Automatically Augmented and Annotated Urban Datasets Using Mixed Reality

Daniel Gómez Casañ

Abstract– This project addresses the topic of database augmentation through a focus on process automation. In the field of autonomous driving systems the datasets used by the learning algorithms are decisive. Furthermore, the underlying machine learning systems would always benefit from having further quality data to learn from. A recent work on the topic of image dataset augmentation has been published, but it did not focus on the automation of the process, and it only involved the addition of cars to the existing images. On the other hand, our project has been developed to also support other kinds of objects. Moreover, our work has centered on developing an automatic pipeline that enables a continual augmentation of the dataset. Thanks to the efforts invested into the analysis of the source images and the automated rendering of virtual objects we can now produce augmented versions of the source images with relative ease.

Keywords– Computer Vision, Autonomous Driving, Dataset Augmentation, Mixed Reality.

1 Introduction

As the fields of Machine Learning and Computer Vision continue to rise in relevance, so does the need for big, well annotated datasets. The amount of samples available for training, validation and testing has a very strong impact on the effectiveness of machine learning systems, and for this reason, a method to automatically augment an available dataset could prove very useful.

In this project, we have developed a solution for automated dataset augmentation. Our proposal consists of a pipeline (as seen in figure 1) with three main modules: input image analysis, object rendering and image composition (also named Synthetizen Compose). The main purpose of the pipeline is the automation factor. Our objective was to be able to generate variations of existing image datasets with as much visual fidelity as the time and techniques used allowed, in order to enhance the results of autonomous driving algorithms.

In the first step of the pipeline, the original images are read and analysed. From that information, an approximation of the 3D scene is generated and then the program decides where the synthetic object will be placed. Using this placement, the second step renders the images which the third step will need in order to compose the real and synthetic parts.

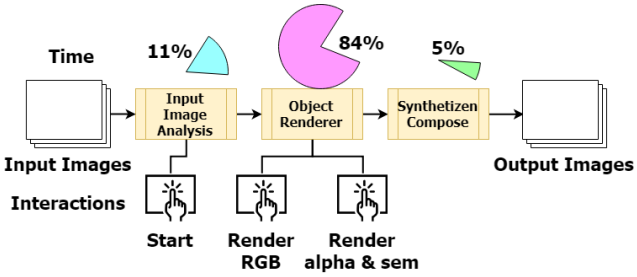

Fig. 1: Summary of the developed pipeline. For a full view see figure 11, in the appendix.

Cityscapes [12] is a large-scale dataset that contains a diverse set of sequences of stereo images captured in street scenes from 50 different German cities, with high quality pixel-level annotations in 5000 frames and 20000 weakly annotated ones. It is very widespread in the field of Computer Vision to develop Autonomous Driving. There are many different types of data in Cityscapes for a given frame. In this project four of those types are used. Three of them are images: left RGB, semantic segmentation ground truth and stereo disparity (see figure 2). And the other one is the file with the camera parameters. Since all four of these files give information about a single frame of the video, they will

• Contact email: daniel.gomezcas@e-campus.uab.cat
• Specialisation in: Computation
• Project tutored by: Antonio Manuel López Peña (Computer Science Department)
• School Year 2017/18

henceforth be referred to as "image set".

Fig. 2: Example of image set. From left to right: Left stereo image, semantic segmentation, disparity

The rest of the sections are organized as follows. In Section 2, we explain the objective of the project, as well as detail some of the major sub-objectives. In Section 3, we will give a brief overview of the state of the art. In section 4, we will go over the methodology followed for the project, that is, the tools and methods applied during the course of the development. In section 5, the experimental results will be presented and discussed. In section 6 we will talk about the conclusions of the project. After that, there will be three additional sections: the acknowledgements, the references and the appendix.

2 Objective

The general objective of this project is to automate the augmentation of image datasets using mixed reality techniques. This means that the process of analysing an image to later create an adequate virtual 3D scene, rendering it and composing the rendered scene with the original image to create a variation should be carried out automatically. The aforementioned process can, of course, range from having a huge scope to a more manageable one, and this project has focused chiefly on the automation aspect, all the while trying to produce the best results that time and resources allow for each step.

Our strategy included a preliminary consideration of a case study and how it would be carried out manually in order to get a better understanding of the challenges. This conditioned how other objectives and their solutions would come to shape themselves.

The main specific tasks are:

- Parameter extraction: To insert a virtual object into an image in a way in which it looks real, we first need to examine this image so that we can determine how and where it goes, and other environment parameters. To do this, there are several sub-tasks that need to be undertaken and will be explained in the methodology section: depth information analysis, geometrical scene approximation, available space in scene determination, placement decision and parameter storage.
- Scene setting and rendering: Once the details of the virtual object have been decided and saved, it needs to be rendered in a series of manners to enable the composition program to join it with the original image. For this part, the sub-tasks that have to be completed are: setting of necessary configuration, rendering and saving of the images and the correct looping over the image set.
- Composition: Once all the images that will constitute the final result are generated, it is necessary to mix them together correctly. This task is made of two subtasks: correct reading of each type of composition image and running and saving the results of the shader composition.
- Stitch together the processes: This is a task that needs to be advanced as the project progresses. When possible, measures are taken so that the separate modules of the pipeline of which the project consists work as seamlessly as possible. This task's complexity highly depends on the capacity of each of the modules to be called and integrate each other, with Unity proving the most difficult to integrate. Its main sub-tasks are: parameter storage and retrieval and correct looping over the image sets .
- Extra enhancements: Due to the main purpose of the project being the automation of the whole process, some aspects could not be developed in a sophisticated fashion and were left to be upgraded outside of the scope of the project if possible. These extra enhancements were added as the project progressed, and ended up being: the dynamic environmental lighting and a better placement decider.

3 State of the Art

Due to the aforementioned rising interest in having more and bigger datasets to train autonomous car algorithms, there have been projects that tried to address this demand in alternative ways. For instance, some approaches, such as [16] and [13], consist in having a big virtual environment in which to generate the data to use for the training and testing of the learning algorithms.

There are other projects, though, that use mixed reality techniques to augment real image datasets. One of the most relevant articles in relation to this project is [8]. In it, Alhaija et al. proposed an alternative to fully virtually generated worlds by combining real and synthetic data for semantic instance segmentation and object detection by adding virtually rendered cars to real images. They take advantage of the fact that not all aspects of the scene are equally important for learning models, and propose to augment the original images with virtual objects. They use a procedure to augment these images using 3D models of cars and couple of ways of determine the position of the virtual car and the illumination model for it.

However, even with good results, in their work they only use virtual cars for augmentation, and it is not clear what degree of automation, if any, they have achieved.

More generally in the field of mixed reality, there are also existing methods that produce very good (as seen in [14] and [15]) results, but they depend on light probes to capture the environmental lighting, and tend to be indoors. These two considerations are certainly not convenient since our intention is to generate synthetic components in exte-

rior scenes with the information provided by Cityscapes (or any other analogous dataset), which does not include light probes. Nonetheless, the differential rendering technique used in [15] is very interesting for image composition and was chosen as our composition method.

Regarding the state of the art, we have seen how there are previous works that achieved image augmentation using mixed reality. However, they are not centered on the automation of the whole process, which is our main purpose.

4 Methodology

The project has been developed following Kanban [10], which is an agile methodology based on software prototype iteration used for managing the creation of products with an emphasis on continual delivery.

Regarding the technology involved in the development of the project, multiple tools and languages have been involved in the work since the start (see figure 11, and will be explained in the context of the module of the pipeline they belong to. First of all, these tools are: Python 2.7, C#, C++, OpenCV bindings for Python (cv2), Unity 3D, Octane Render for Unity, Qt API, GLSL, assimp bindings for Python (pyassimp), Meshlab and Autodesk Maya.

4.1 Image Set Analysis

The Image Set Analysis is the module that takes care of extracting the parameters for the Object Renderer to set the scene. It consists of the following sub-tasks (grouped in 3 segments in figure 3):

1. Depth information analysis: Using the disparity between the stereo cameras of Cityscapes, we can obtain the coordinate (depth) that is missing from the image.
2. Geometrical scene approximation: To judge the image we first need to approximate it to the 3D scene it captured. Here, we use the previously obtained depth to build the scene approximation that will be crucial for the next steps.
3. Available space in scene determination: To help produce a more reasonable result, the space where the object can be placed in the scene is limited to the surface on which it makes sense to be placed (e.g. car on road).
4. Placement decision, simulation and confirmation: When there is a surface or surfaces where the object could be placed, a placement needs to be decided, and then it needs to pass a test to check if it is intersecting with other objects in the scene.
5. Saving the obtained parameters: To help the overarching goal of joining the different pieces of the pipeline, it is very important that the obtained results in each part are saved for later use.

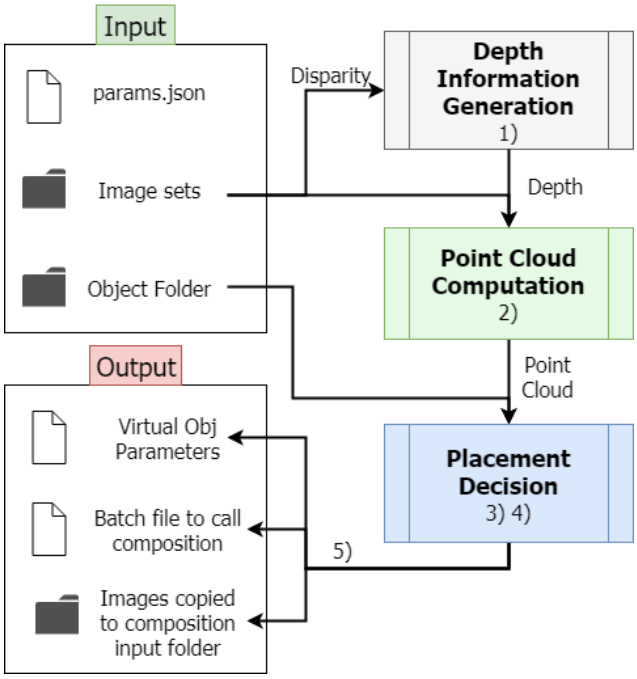

Fig. 3: Tasks of the first module

The first module of the pipeline has been developed with Python 2.7 at its core. It uses the OpenCV [9] wrapper for Python (cv2) to read, operate and debug images, some Cityscapes helper scripts and pyassimp[1] in order to read and interpret the FBX [2] models used for our virtual objects.

This module takes about 11% of the total time that the pipeline is running, it requires manual startup and it's the first one to run.

In this first step of the process, the file with parameters and the source information from Cityscapes is read: leftImg8bit (left stereo RGB image), gtFine (semantic segmentation ground truth), disparity (disparity between stereo images, used to get the depth image) and camera parameters. Once everything has been loaded, the program starts looping over the specified number of image sets for every specified city.

Here is where the important processes take place. For every image set, the depth image is obtained from the camera parameters and disparity information using a very well-known formula (see Eq. (1), with focal length and disparity being in pixels and baseline in meters).

$$Depth = focal\ length * Baseline/disparity \quad (1)$$

With the depth information, the point cloud of the scene represented in the images can be computed and coloured using the RGB or semantic images. To obtain the point cloud, we use the camera parameters and the pinhole camera approximation (see Eq. (2), and figure 12 in the appendix as a visual aid) and convert all the 2D points into 3D ones. In

[1] Python bindings for assimp [4], which in turn is short for Open Asset Import Library and is a portable Open Source library to import various well-known 3D model formats

this equation, u and v are the coordinates of the projected 2D point; X, Y and Z are the coordinates of a 3D point in the world coordinate space; the intrinsic matrix describes the geometrical properties of the camera; and finally, the extrinsic matrix describes the motion of the camera around a static scene.

$$s\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{Intrinsic\ Matrix} \underbrace{\begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix}}_{Extrinsic\ Matrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{2}$$

To debug the point cloud sub-systems, Meshlab [11] is used. It is an open source system for processing and editing 3D meshes, and it lets us easily see how the point cloud is coming out and, later on, where the bounding box of the virtual object is in respect to the rest of the points.

Once the point cloud is ready (similar to what can be seen in figure 4), the planning of the object's positioning in the scene can begin. The current method we use for this is a simplistic approximation but also effective enough to produce good placements consistently. To get a matrix with the points where the object could be placed, the type of virtual object to be used is passed to a dictionary with the labels of where it can be placed (e.g. car on road, person on sidewalk).

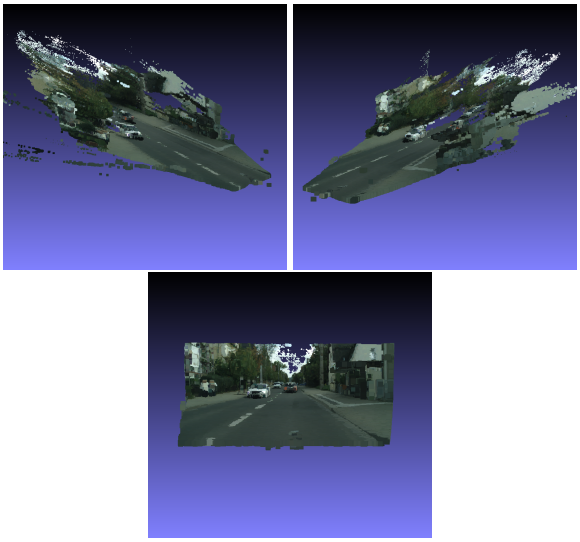

Fig. 4: Three perspectives on the point cloud of a Cityscapes image

Afterwards, all points with labels other than those allowed are discarded, and a random allowed point is chosen. However, this could cause obvious intersection problems, and so the placement function checks for intersections in order to validate the placement. The function that checks for intersections has a strict method and a lenient one. The strict method checks the bounding box of the car against all points, using vector operations. Given the potentially huge number of points in the point cloud, the strict check could be very slow. To circumvent this issue a fast pre-filter is ran first using the lenient method. The pre-filter uses a couple of simple comparisons that might give a false positive intersection in case something is too close. If no intersections are detected, a lot of time has been saved, and if they are, the strict check is executed only on the suspected points. To see the graph comparing pre-filter times on all points vs. strict filter times on pre-filtered points, see figure 13. This verification process is executed until a placement appears to be correct. It is necessary to disclaim that some times, erroneous placements are generated. However, these are mostly due to inaccurate depth estimations in the point cloud and not to undetected intersections.

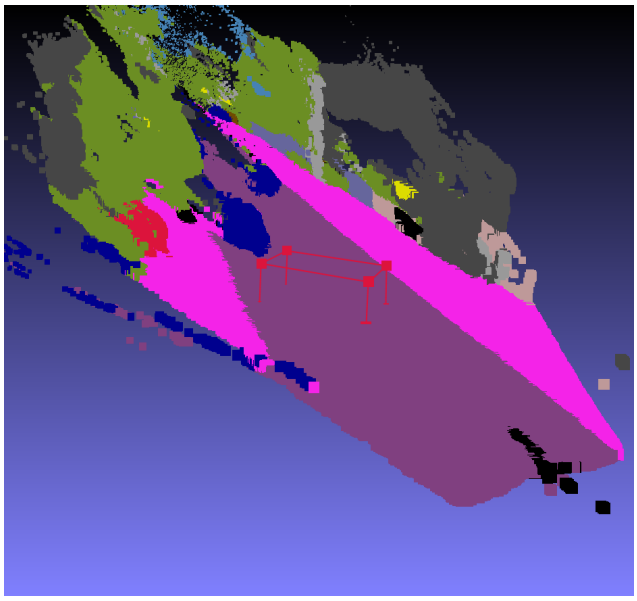

Fig. 5: Example bounding box position with a given orientation

However, there can be problems with the virtual object's model. For these cases, as well as for debugging the intersection checkers and the point cloud in general we also use Meshlab [11]. With it, we can see the point cloud and where the bounding box of the object lies within it (as shown in figure 5). For trouble concerning the coordinates of the edges of the bounding box, which involves the frame of reference of the model's file, we use Autodesk Maya [1]. This tool, however, has only been needed twice, to make small fixes to some models.

Lastly, when the placement for the current image set has been decided, all that is left are a couple of operations. Firstly, the placement of the object and the rest of parameters is saved in a file for the second module (Object Rendering) to read them. Afterwards, a copy of the source images is made in the input folder of the third module (Synthetizen Compose). Lastly, the necessary command for the current image set to be composed is added to the execution batch for the third module (its execution is explained in section 4, subsection 3).

4.2 Object Rendering

Object Rendering is the module in which virtual scene parameters are read, and the virtual images are generated. It consists of the following sub-tasks:

- Setting of necessary configuration: Using the Unity 3D [7] engine, the virtual elements need to be placed as specified by the parameters that were computed in the previous stage.
- Rendering and saving of the images: Using the Octane Render [3] plugin for Unity 3D, the necessary images are rendered for later usage.
- Looping over every image set: This step is mentioned explicitly since this module has to be integrated into the Unity state machine. Also, multiple renders per scene are needed and this objective comprises the creation of a custom state machine to handle the transitions between the different renders and scenes.

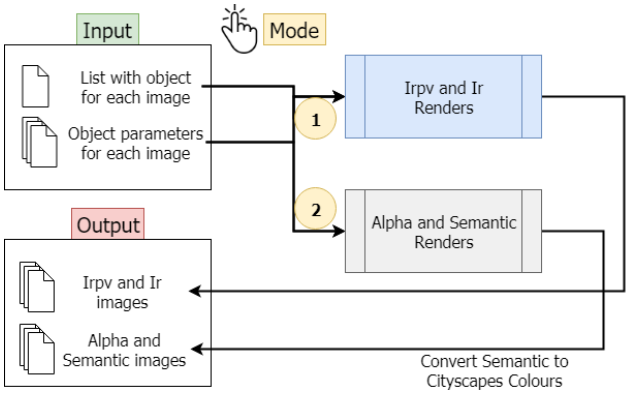

Fig. 6: Diagram of the second module

The second module of the pipeline was developed using Unity 3D [7], with a specific plugin to run the rendering (using Physically Based rendering) called Octane Render [3], which produces high quality dynamic range images. The code for this module was developed with C# and the time it took until completion was heavily impacted by the fact that there is no documentation for the plugin. Moreover, the internal settings of the plugin's render target, which controls all of its options, are not accessible programmatically and this added an additional required manual interaction as well as making some project objectives unattainable in the way they had been conceived (namely, the per-scene illumination simulation using the plugin's environment lighting). However, we also found that we could generate semantic segmentation ground truth using this plugin, and did so in the late stages of the project as an important added feature.

This module takes about 84% of the total time, composed of 75% for the RGB images and 9% for the alpha and semantic ones. It requires manual startup when initiating either render modes (explained below).

There are two essential steps (using different Octane kernels) to rendering the necessary images: RGB and Opacity+Semantic. The order is interchangeable as long as the checkbox for last render bunch is checked when rendering the second group of images. This is because it will trigger the third module on finish, eliminating the need to do so manually.

When execution starts, Unity's Start function is called. There, parameters are loaded and conditions, checked. Depending on which kernel is active (indicated by the user using the Unity inspector), a different set of elements is activated and deactivated in the scene. To allow the meshes to be loaded, a separate thread blocks the first one every time new parameters are loaded. Essentially, though, the program will keep rendering and saving the required images until it is done. The RGB segment of this module is very much slower than the other one because it renders the full object in RGB and after that it renders the virtual surface on which it is standing. In comparison, when the other segment runs, it only needs to render the opacity component of the object while also saving a render pass with the Render ID of the object (which will be later converted to Cityscapes colour code).

To get into detail, the RGB segment loads both the object and the surface, waits for the desired samples per pixel of the render to be reached, saves the image and changes to rendering only the surface for the same configuration. When it is done, it increases the counter and starts over for the next scene.

With the Alpha+Semantic segment, something similar occurs, except that instead of having to change from one render to the other, both are done at the same time, since the "semantic segmentation" is in a different Render pass than the Opacity. During the development of the project we intended to render the opacity and semantic segmentation at the same time than the RGB images (saving one manual interaction), but the generated Opacity image was different than the one that we obtained by changing the kernel (which required manual interaction). The lack of documentation made things slower and when we got the third module to work with the new opacity images it was observed that these did not have an alpha component. This caused the composition to come out jagged and the changes had to be discarded. Despite these drawbacks, the rendering of the alpha (Opacity) and semantic images is several times faster than the RGB renders due to their simplicity.

For the semantic segmentation image, the colours need to be adjusted to the Cityscapes values, so in its case, instead of just being saved to disk, the pixel buffer is read, cloned with the necessary changes, encoded as PNG and saved as bytes.

In the end, for each scene, four images are generated: I_{rpv} has the object and the surface below it, I_r has only the surface, alpha is the opacity of the object alone and semantic is the shape of the object with the colour corresponding to its type (Cityscapes colours). A sample of the four can be seen in figure 7.

4.3 Synthetizen Compose

The Synthetizen Compose module is the one in charge of composing the generated images with the real ones. It consists of the following sub-tasks:

- Reading involved images: This module is carried away using shaders in a Qt [6] environment and thus

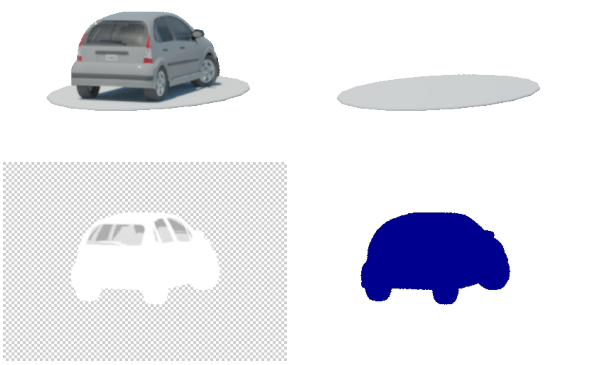

Fig. 7: From top to bottom, left to right: Irpv, Ir, Alpha, semantic

the step involving image reading is crucial for the module to work.

- Running shader composition and saving: The images needed to generate the composition are ran through a shader, and the buffer is saved as the output image.

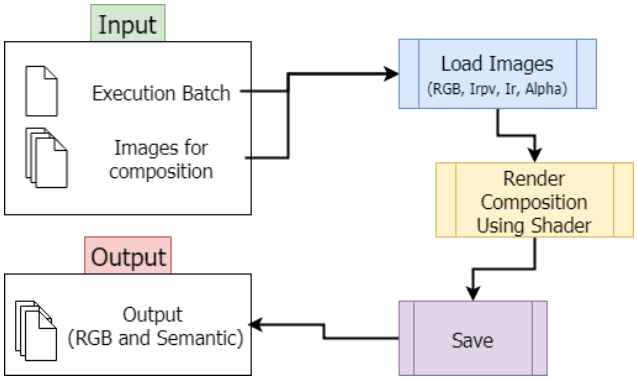

Fig. 8: Tasks of the third module

The third module of the pipeline is the composition program. It has been developed using C++ in a Qt [6] application that uses shaders written in GLSL [5] to compose the pieces that will make up the final image. The method used for composition is known as "differential rendering". Differential rendering is a standard technique for simulating the light transport in mixed reality rendering. It operates on pixels from different images to join them, using the formula seen in Eq. (3). In this formula, we find that R (the original image) becomes R′ by joining I_{rpv} and R + I_{rpv} - I_r using α (also known as opacity) as a mask of sorts. I_{rpv} is the rendered image of the virtual object on a virtual surface and I_r is the rendered image of only the virtual surface.

$$R' = \alpha * I_{rpv} + (1 - \alpha)(R + I_{rpv} - I_r) \quad (3)$$

This module only takes about 5% of the time and runs automatically after the third, since all of its preparations have been carried out beforehand. As mentioned previously, the first module writes the batch with the commands for the compositions of all the image sets and copies the images that the third needs to its source folder. So when its time comes, everything is set.

For every image set, this module loads the different components of the differential render (I_{rpv}, I_r, opacity and background) and uses the shader to create the final image, pixel per pixel. In the case of the semantic composition, the I_r image is ignored since we only need the new object and not the shadows and reflections.

4.4 Stitching together and enhancements

Finally, the different modules had to be stitched together. This mostly had been prepared gradually during the development of each module. However, it is important to point out that many procedures had to be adapted to set up the necessary files for other modules and join them as seamlessly as possible.

This mainly included the following sub-tasks:

- Parameter retrieval: The parameters for a given module need to be loaded from a file that a previous module generated.
- Correct looping: Due to the nature of some parts of the pipeline, the looping is not always straightforward, and specifically for the second module, a state machine was developed in order to correctly handle the iteration over the different image sets.

Additionally, if time allowed it, we wanted to develop more sophisticate methods to improve the results. The following enhancements were planned:

- Environmental lighting: An interesting feature to add would be to use the source image to illuminate the scene that is generated in the render engine.
- Semantic ground truth generation for the new images: The generation of semantic segmentation ground truth was added further on due to its impact on the usefulness of the project.

The environmental lighting couldn't be developed as planned because of limitations in the functionalities of the Octane Render plugin at the time. However, a sophisticated method was theorised that might have been able to act as a substitute.

As for the semantic ground truth generation, we managed to add it to the segment of the second module in which the alpha image was being generated. The process to get the semantic ground truth is simple but required changes in the input parameters too. An ID is assigned to the virtual objects to be composed and a render pass is assigned with the task of rendering the ID as a colour. However, since the colour is not the one Cityscapes uses, what happens is that the program reads the pixel buffer of the render pass and copies it to a Unity texture, changing the pixels that correspond to the object to the correct Cityscapes color code for that type of object.

5 EXPERIMENTAL RESULTS

The results of this project chiefly consist in the output images. They must be valued according to how valid they are as new images to be fed into an autonomous driving learning agent. However, this is not the only metric by which the results of the project are evaluated. If the pipeline is to be rendering new data, it can not take large amounts of time in doing so.

5.1 Evaluation metrics

As it has just been briefly mentioned, there are various aspects to take into consideration when analysing the results:

Amount of required interaction: Since the main objective of the project is to automate the process from image analysis up to image composition, it is necessary that the automation that has been reached is taken into consideration when evaluating the results. Aside from turning on the pipeline, there are two other times when the pipeline requires user input. The first one is after the placement has been decided and the Object Renderer needs to be ran, and the second one is when the render mode needs to be changed from RGB to alpha and semantic or vice versa.

Perceived subjective quality: The correctness of the assigned position of a virtual object is not taken into account within this aspect due to the fact that the project does not focus on the placement algorithm and the object is just assigned to be placed anywhere inside the allowed boundaries. However, there are other factors that weigh in this aspect. As has been mentioned in the methodology, the current state of the Octane plugin has not allowed us to tinker with the illumination as would be needed to adapt to each scene. For this reason a general illumination has been chosen considering the usual weather observed in the Cityscapes images.

The quality of the results is a metric that has been used more to observe improvement over the iterations of the systems than to compare outputs within a given iteration.

Execution time: Despite not being the main objective of this project to get the fastest possible results, the execution speed of the pipeline has always been taken into consideration as a relevant parameter. Several adjustments such as changing the point cloud to be calculated matricially using numpy or the pre-filter/strict filter system of the intersection check were developed with speed in mind. The approximate relative duration of each module in respect to the total time elapsed for a batch of images is represented in figure 11.

5.2 Final Results

During the development of the project we have worked on augmenting a small set of about 18 images in order to quickly debug and iterate over a familiar set of data. Thus, figure 9 shows several representative examples in which, using the developed pipeline, either a car or a traffic sign are added to the scene. For more examples of results, see figure 15 in the appendix.

A graph depicting the times used in every module as the number of image sets increases is given in figure 10. The total time increases linearly with the number of image sets, as suggested by the coefficient of determination of the trend line resulting from this plot.

5.3 Discussion

The following results have been produced in a 64 bit Windows OS machine with 24.0 GB of RAM and running on an Intel® Xeon® CPU W3565 at 3.20GHz (3193 Mhz, with four cores, eight logical processors). For graphics it has a NVIDIA GeForce GTX 680 Graphics Card.

Given that the Cityscapes training, testing and validation folders sum up 5000 image sets, we can safely estimate that it would take approximately 2,265,461 seconds (26 days, 5 hours, 17 minutes and 41 seconds) to augment the whole Cityscapes dataset, which is huge. Even though this may seem too big a number, it is not so unreasonable since it only needs to be ran once and it is for the whole dataset. Furthermore, these numbers only consider that only one machine like the aforementioned one is being used. If a more powerful machine were to be used, the times would surely be reduced (with the graphics card being the most important component, since the majority of the time is used up in the renders). Moreover, since the process is highly parallelisable and the time grows lineally with the amount of images, every additional machine running the augmentation would greatly reduce the needed time.

Concerning the final quality of the results, it should be mentioned that the illumination could not be accessed using Octane at the time this project was developed. This was unfortunate because it appears as the only problem once you have the object in a good placement. In an attempt to solve this adverse circumstance we used a dimmed out daylight environment for illumination (since most of Cityscapes images have cloudy scenes), to try and approximate a good lighting for as many images as possible.

In the case of the semantic segmentation augmentation, it can be seen how the generated images resemble the original greatly (except for the different tone mapping caused by the HDR). The colour is the same as the ones from the source dataset and the images are spatially coherent. In order to make the semantic segmentation augmentation better, work would have to be put into other parts of the pipeline, and also into producing the output in another image format, or with a custom tone mapping to match the colours in Cityscapes.

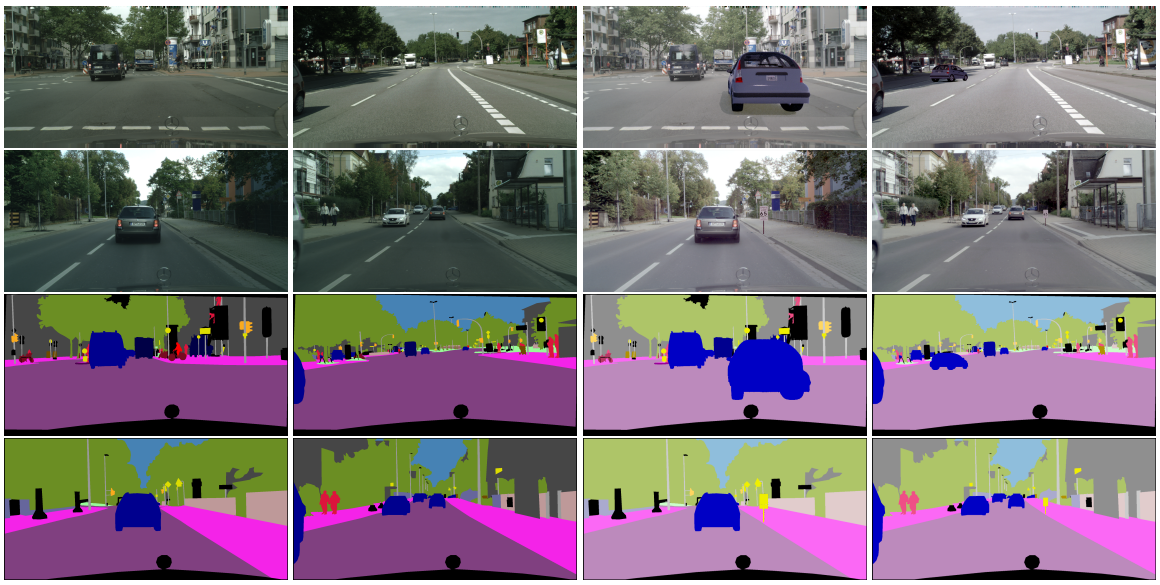

Fig. 9: Examples of source images (first two columns) versus their augmented counterparts: RGB images (top, two with added cars and two with added traffic signals) and their respective augmented semantic segmentation images (bottom)

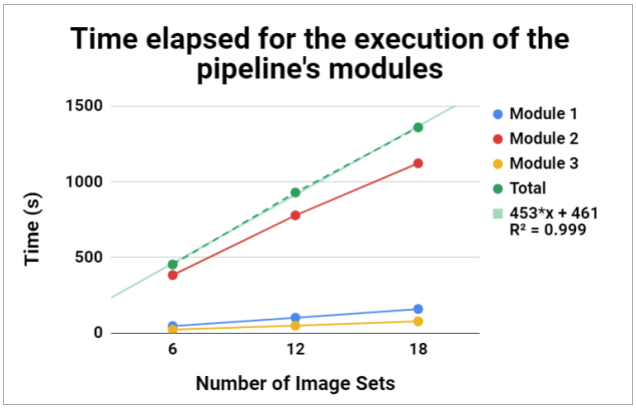

Fig. 10: Times for every module for 6, 12 and 18 image sets (raw data in figure 14)

6 Conclusions

We have been able to build an automatic pipeline for the augmentation of image datasets. Thus, in conclusion we can state that the project's objectives have been attained successfully.

For a simplistic approximation of the scene, the results can be deemed accurate, specially the semantic segmentation ground truth, which is indistinguishable from the one in Cityscapes.

Overall, the process of automation has been very satisfactory. Moreover, despite the automation requiring a pair of human interactions, since 75% of the time is spent rendering the RGB images, it can be concluded that for the vast majority of the process, no interaction is required.

Regarding future upgrades and enhancements, there are several approaches that would greatly improve the quality of the RGB output, and the project in general:

- First of all, while the current render engine doesn't allow the parameters to be tinkered with and edited from the code, we would have to study alternative ways of having a dynamic lighting. For instance, the illumination could be approximated using point lights around the virtual object in the Unity scene.

- In the line of what was done with the semantic segmentation data, the depth information could be augmented similarly. Despite the fact that it is in the form of stereo disparity in Cityscapes, it could be normalised with the depth information generated in one of Octane's render passes to create a new set of data for the dataset, which would essentially be the evolution of its disparity data.

- Next up in priority would be the development of a sophisticated algorithm to determine the placement of objects in the scene. This would entail the already existing label restrictions, and also probably computing the homography of the scene to examine road lines and other contours and extract the best placement and orientation for any given type of object.

- To aid in the process of deciding which virtual object to add to a given scene, a method that involved the existing knowledge on the frequency of different types of instances should also be studied and developed.

- A way to eliminate the issues with the missing alpha channel in Octane's opacity render pass would be very useful to the pipeline, since it would make it almost fully automatic. Or if that was not possible, we might be able to find a way to generate it ourselves from the information present in the render passes.

- As an extra enhancement, the way to increase the speed of the RGB rendering should also be studied in order to reduce the bottleneck effect it has on the pipeline.

ACKNOWLEDGEMENTS

First of all I want to thank my director, Dr. Antonio M. López Peña and Dr. José A. Iglesias Guitián for their help and support throughout the project. The work I have done would not have been possible without everything they have taught me and the help they have provided when I needed it. They have always been willing to offer a helping hand and I really appreciate it.

I would also like to thank my colleagues and coworkers for taking me in as a member of the team. Their support and the cheerful work environment have driven me to do my best. And their assistance has proven invaluable to understand many concepts and techniques.

I also wish to thank my family. I am deeply grateful for their support and effort to help me finish my education.

And last, but not least, I would like to thank my friends for their support and company. Without them, my stay in college would have had much less laughter and lasting memories.

REFERENCES

[1] Autodesk maya. https://knowledge.autodesk.com/support/maya. Retrieved: 2018.06.26.

[2] Filmbox - 3d file format. http://docs.autodesk.com/FBX/2014/ENU/FBX-SDK-Documentation/. Retrieved: 2018.04.22.

[3] Octanerender for unity. https://home.otoy.com/render/octane-render/. Retrieved: 2018.04.22.

[4] Open asset import library. https://github.com/assimp/assimp. Retrieved: 2018.04.22.

[5] Opengl shading language. https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language. Retrieved: 2018.06.26.

[6] Qt - cross-platform sdk. https://www.qt.io/. Retrieved: 2018.06.26.

[7] Unity 3d game engine. https://unity3d.com/. Retrieved: 2018.04.22.

[8] Hassan Abu Alhaija, Siva Karthik Mustikovela, Lars M. Mescheder, Andreas Geiger, and Carsten Rother. Augmented reality meets computer vision : Efficient data generation for urban driving scenes. *CoRR*, abs/1708.01566, 2017.

[9] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[10] Eric Brechner. *Agile Project Management with Kanban*. Microsoft Press, Redmond, WA, USA, 1st edition, 2015.

[11] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In Vittorio Scarano, Rosario De Chiara, and Ugo Erra, editors, *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008.

[12] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. *CoRR*, abs/1604.01685, 2016.

[13] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[14] Andrew Gardner Ehsan Miandji Jonas Unger Joel Kronander, Francesco Banterle. Photorealistic rendering of mixed reality scenes. *Computer graphics forum*, 34(2):643–665, 2015.

[15] Kevin Karsch, Kalyan Sunkavalli, Sunil Hadap, Nathan Carr, Hailin Jin, Rafael Fonte, Michael Sittig, and David Forsyth. Automatic scene inference for 3d object compositing. *ACM Trans. Graph.*, 33(3):32:1–32:15, June 2014.

[16] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M. Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

APPENDIX

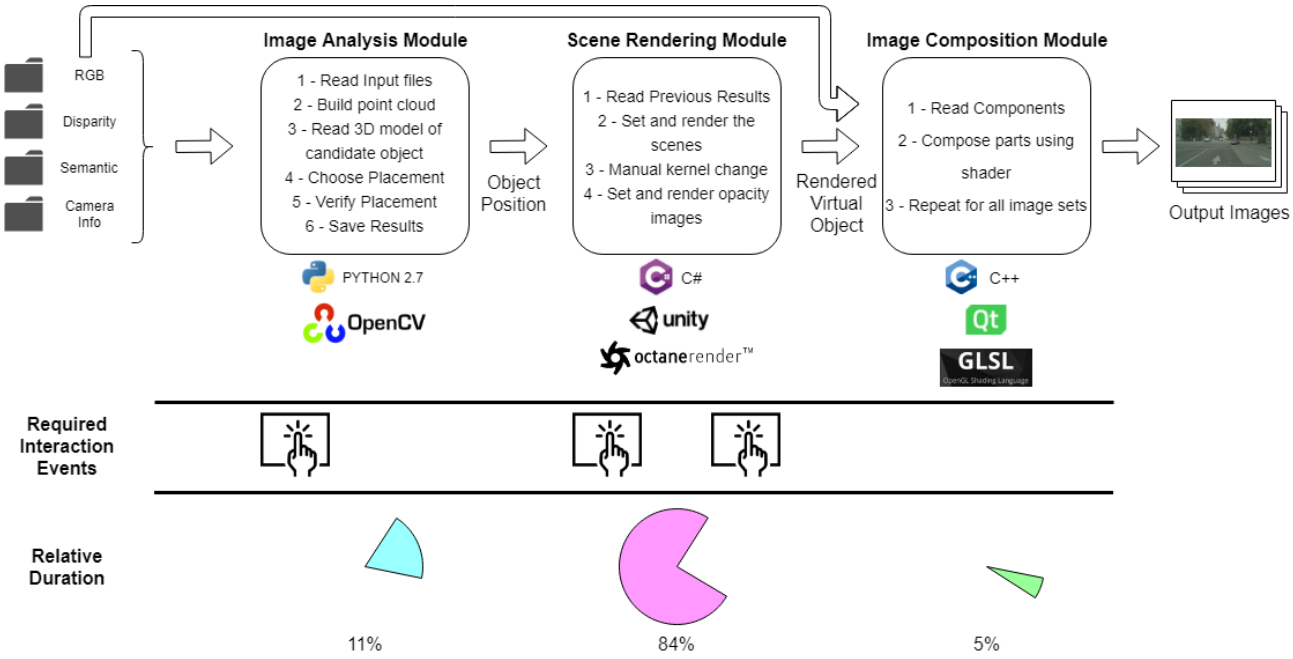

Fig. 11: General overview of the pipeline

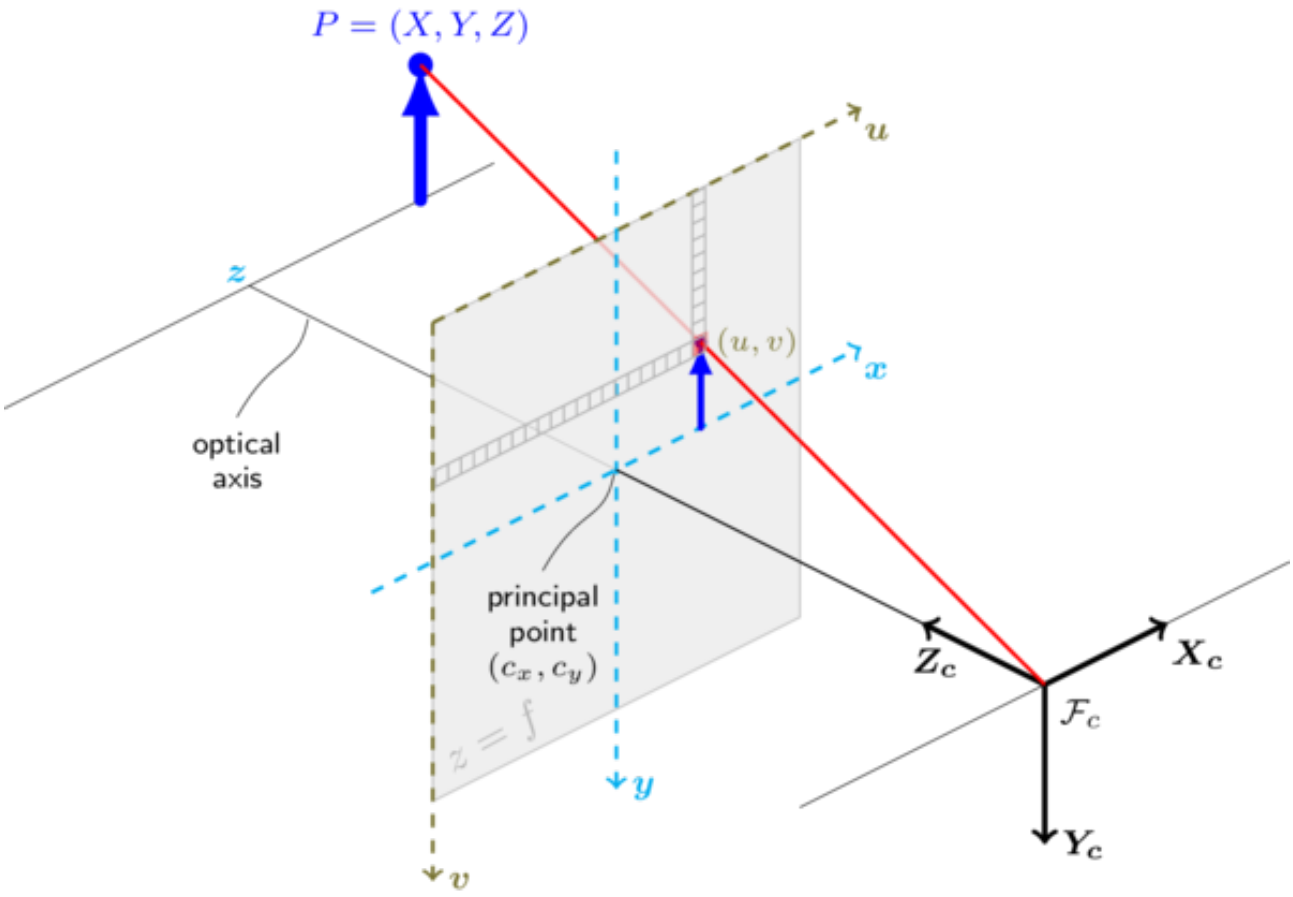

Fig. 12: Diagram of the pinhole camera model

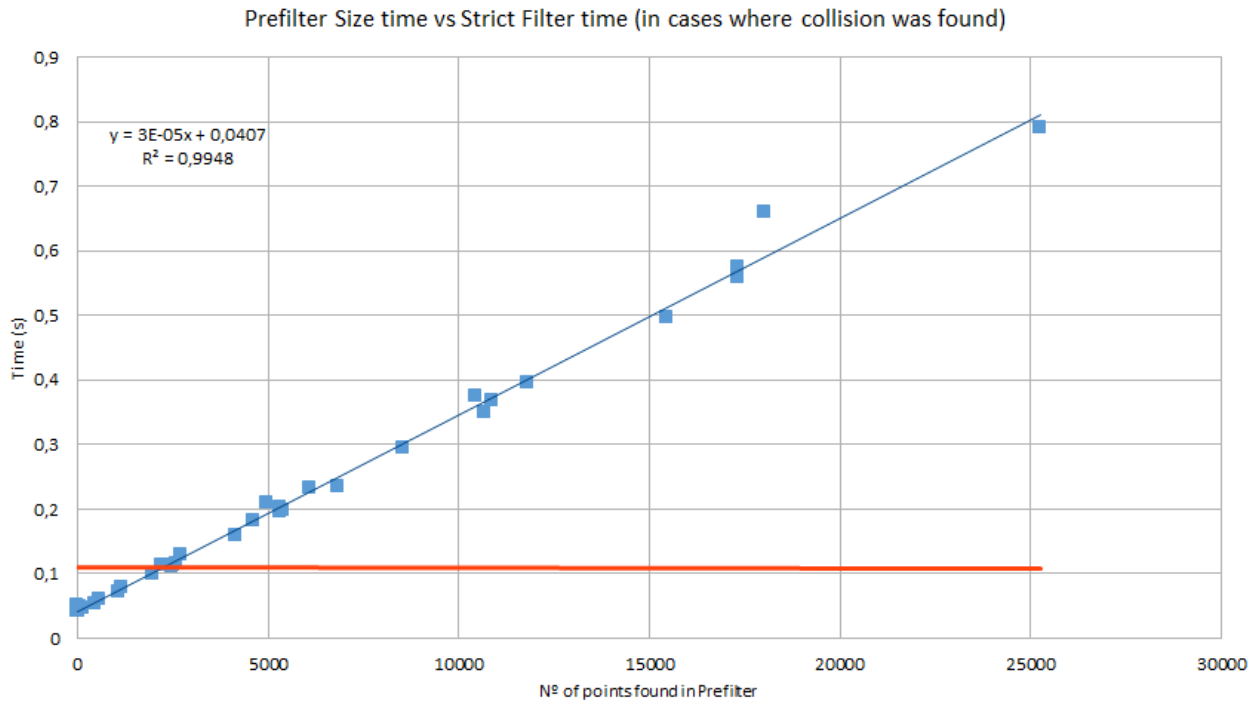

Fig. 13: Time for prefilter on 1M points vs time for strict filter of remaining ones

	Input Image Analysis		Object Renderer			Synthetizen Compose		
# of Image Sets	Time (s)	%	Irpv&Ir (s)	alpha & sem. (s)	%	Time (s)	%	Total (s)
6	46.75	10.32	337.28	45.78	84.56	23.22	5.13	453.03
12	101.81	10.97	695.70	82.11	83.78	48.82	5.26	928.44
18	159.15	11.72	1,014.06	107.68	82.58	77.52	5.71	1,358.41

Fig. 14: Times and % for every module for 6, 12 and 18 image sets

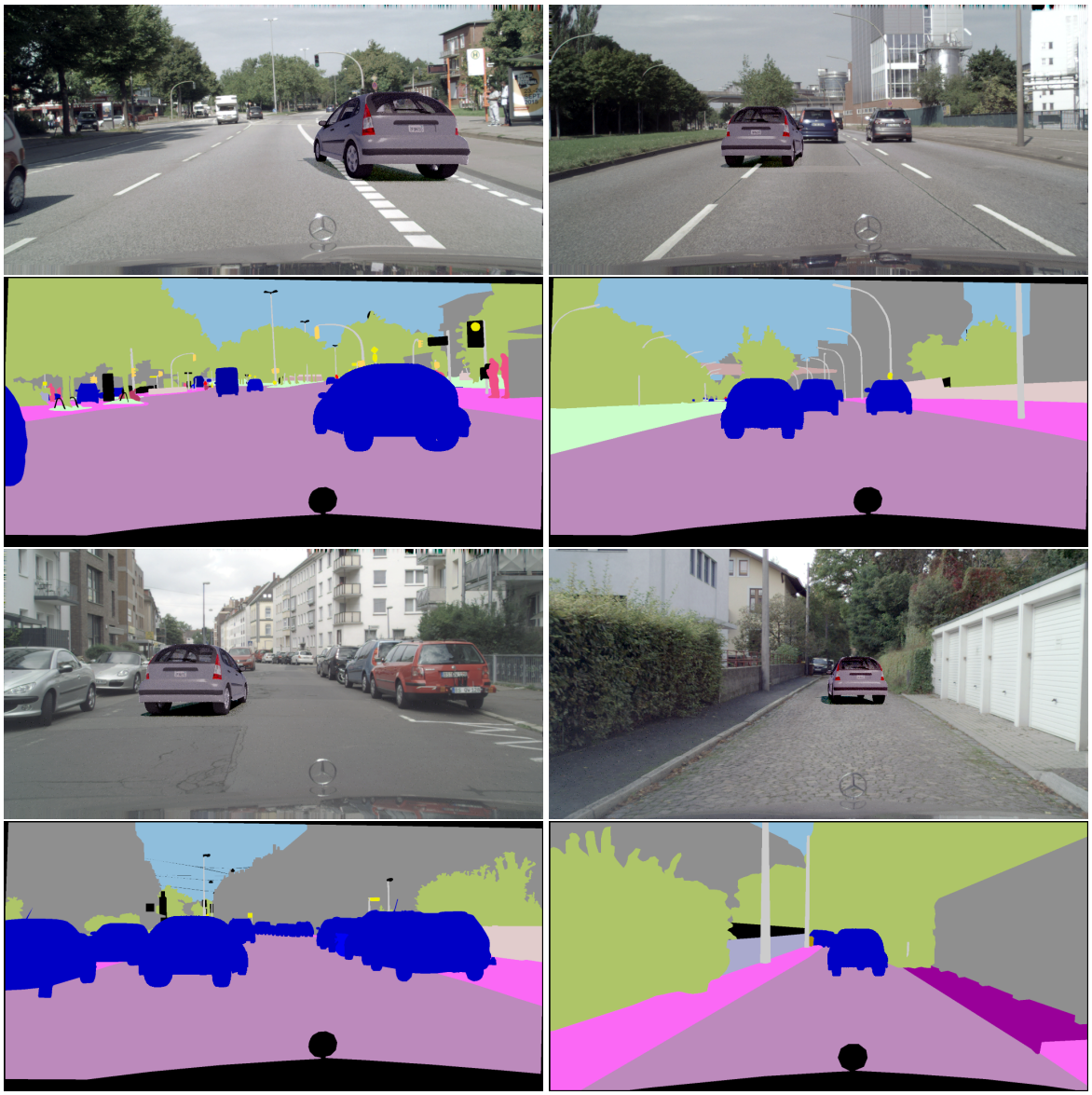

Fig. 15: Additional examples of augmented RGB images. For every pair of images: top is augmented RGB and bottom is augmented semantic